

# Garbage Collection for Prolog Based on WAM

KAREN APPLEBY, MATS CARLSSON, SEIF HARIDI, and DAN SAHLIN

**ABSTRACT:** The Warren abstract machine (WAM) has become a generally accepted standard Prolog implementation technique. Garbage collection is an important aspect in the implementation of any Prolog system. A synopsis of the WAM is presented and then marking and compaction algorithms are shown that take advantage of WAM's unique use of the data areas. Marking and compaction are performed on both the heap and the trail; both use pointer reversal techniques, which obviate the need for extra stack space. However, two bits for every pointer on the heap are reserved for the garbage collection algorithm. The algorithm can work on segments of the heap, which may lead to a significant reduction of the total garbage collection time. The time of the algorithms are linear in the size of the areas.

## INTRODUCTION

A variety of techniques, such as tail recursion optimization, have been developed in an effort to minimize memory consumption. However, none of these techniques have reduced the amount of garbage enough to reduce the role of garbage collection in Prolog. In developing an algorithm, one must keep in mind that the heap is used in a stack fashion; it grows during forward execution and is unwound during backtracking. This requires a garbage collection algorithm that maintains the order of data created on the heap. The main algo-

rithms in this paper are linear, requiring two bits per word of pointer storage. The ideas of segmented garbage collection [8, 13] are incorporated in the design. The algorithms are for use by a single processor system.

The rest of the article is organized as follows: First a synopsis of Warren's Prolog implementation, the Warren abstract machine (WAM), is given. Then a general marking and compaction algorithm is described. As a first optimization of the algorithm, it is shown how to use the trail information to increase the amount of detectable garbage and how to compact the trail. Finally, segmented garbage collection, a method for reducing the search space, is discussed.

## SYNOPSIS OF THE WAM

In 1983, Warren published a description of an abstract machine for Prolog execution [16]. The paper describes an instruction set and several data areas that the instructions operate on. The instructions are oriented toward the source language rather than toward the target machine. Thus an abstract instruction usually corresponds to a source program symbol, but may involve a large, even unbounded, number of machine operations.

Warren's instruction set has since become generally accepted as the standard Prolog implementation technique and has been realized both by bytecode emulators, compilation into native instructions, and emulation in firmware and hardware. We do not contemplate these techniques further.

Best Available Copy

Several authors have described optimized or extended versions of the WAM. This paper contains a condensed description of a somewhat simplified WAM. For instance, we do not treat all instructions, nor their complete semantics. A more detailed description of WAM may be found in Gabriel [7] and Warren [16].

### Data Areas

The data areas are the *code area*, containing the program itself, the *control area*, containing the abstract machine registers, and three areas operated as stacks. The (*local*) *stack* contains information pertaining to backtracking and recursive procedure invocations. The *global stack* or *heap* contains structures, lists, and value cells created by Prolog execution. The *trail stack* contains references to conditionally bound variables, that is, it records such bindings that have to be undone upon backtracking. (See Figure 1.) The stack is assumed to grow in the positive direction, toward higher addresses, which is downward in our figures. The first cell of the heap, the stack and the trail are called *heap\_low*, *stack\_low*, and *trail\_low*, respectively.

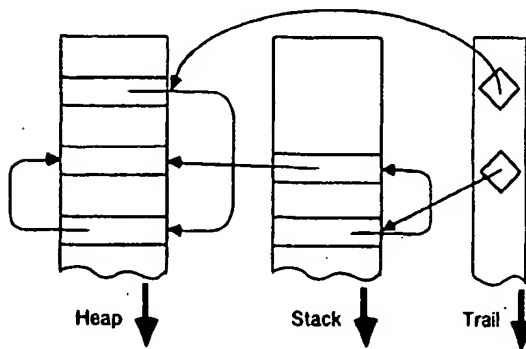


FIGURE 1. Three Data Areas Operated as Stacks

### The Heap

An important concept is that of a *value cell*. A value cell stores a Prolog term and consists of a *tag*, a *value*, and two *bits* used in garbage collection. The heap consists exclusively of value cells. There are value cells on the stack as well. The following rules govern how value cells may refer to one another.

- (1) A heap cell may only refer to another heap cell.
- (2) A stack cell may refer to an earlier stack cell or to a heap cell.

A possible pseudo-C declaration for value cells could be

```
struct valuecell {
    int tag;
    struct valuecell *value;
    bool m;
    bool f;
};
```

During the marking phase, the *m* bit is set for the marked value cells, whereas the *f* bit distinguishes the first cell of a structure or list. Both bits are initially FALSE. The tag distinguishes the type of the term, the main types being references (VAR), structures (STRUCT), lists (LIST), and constants (CONST). An unbound variable is represented as a reference to itself. A value cell tagged as a variable but not pointing to itself represents a bound variable. Structures are created by explicitly copying the functor and arguments into consecutive value cells on the heap. The value part of the functor value cell encodes its function symbol *f* and arity *n*. Functors are written as *f/n*. Lists are created similarly, except that no functor needs to be stored. This technique for creating structures and lists is known as *structure copying*. For constants, the value field refers to something outside the scope of garbage collection; that is, the cell either points to a static area or contains a static value, for example, an integer. In any case, the contents of the cell are ignored by garbage collection. We depict the different types of terms as shown in Figure 2. Note that a variable may point at an individual element of a list or a structure, but may not point at the first component of a structure, that is, at the functor.

### The Control Area

This area consists of pointers into the other areas plus a scratchpad register bank for passing arguments and for temporary usage. The contents of the control area registers define the computation state. In the rest of this paper, we treat the control area registers as if they were globally accessible machine registers. Indeed, that is how they are typically implemented.

```
struct wam {
    struct code *P;
    /* program pointer */
    struct code *L;
    /* continuation program pointer */
    struct environment *E;
    /* current environment */
    struct wam *B;
    /* current choice point */
    struct valuecell **T;
    /* top of trail stack */
    struct valuecell *H;
    /* top of heap */
    struct valuecell A[m];
    /* m argument registers */
};
```

where *m* is a suitable number defining the arity limit of functors and predicates.

A few words about C syntax and semantics: The declaration `struct wam *B;` means that *B* is a pointer to a saved WAM state. Since *B* is such a pointer, then `B → P` is the *P* component of that structure. The declaration `struct valuecell **T;` means that *T* is a pointer to a pointer to a value cell. Dereferencing one step is done by `*T`, which is just a pointer to a value cell.

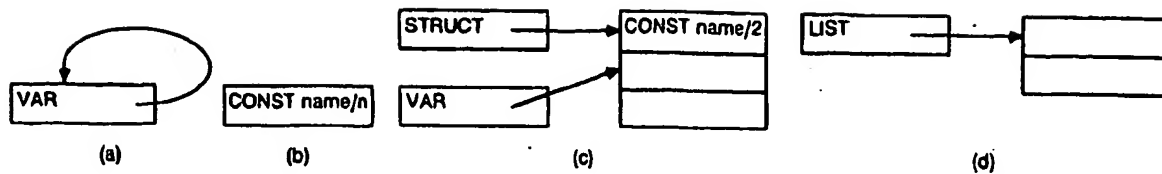


FIGURE 2. Different Types of Terms. (a) An Unbound Variable. (b) A Constant with Arity  $n$ . (c) A Structure and a Variable Bound to a Component of the Structure. (d) A List Cell Pointing to the Head Cell.

There are two more items that are used over short sequences of instructions.

```
struct valuecell *S;
/* current structure */
bool RW:1;
/* read or write mode bit */
```

However, these are never valid when garbage collection occurs and so are not included in the struct wam definition.

### The Local Stack

The local stack contains two kinds of structures: *environments* and *choice points*. An environment represents a list of goals still to be executed. It consists of a number  $k$  of local variables occurring in the body of a clause plus a pointer into the body of a continuation clause and its environment. A local variable may be unbound, in which case it points to itself just like unbound heap variables. Unbound variables on the stack save heap storage but complicate the instruction set somewhat. We ignore these complications in this description.

Environments are only needed for clauses with more than one body goal and are established when entering such clauses. They are (logically) discarded before executing the last body goal. The format of an environment is

```
struct environment {
    struct code *CL;
    /* continuation program pointer */
    struct environment *CE;
    /* continuation environment */
    struct valuecell Y[k];
};
```

An environment can be *active* or *inactive*. An active environment is in the current execution path, in which case it is in the chain formed by the CE fields. An inactive environment is one that is associated with a clause that has finished execution, but may be reactivated upon backtracking. In this case it is only reachable via a choice point. Since an environment refers to its parent environment and several environments may have the same parent, the environments form a tree which has one active leaf,  $E$ , and zero or more inactive leaves.

Figure 3 shows the tree structure of the stack and the trail. A rectangle denotes an environment, whereas a

rounded rectangle denotes a choice point. A diamond denotes a trail cell, which in turn points either to the stack or to the heap (not shown in the figure).

A choice point is established when entering a procedure  $Q$  with  $n$  arguments that has more than one clause that could match the goal. It consists of a snapshot of the control area. Thus the format of a choice point is a struct wam, except the argument registers  $A$  contain a copy of the  $n$  arguments of  $Q$ , and the  $P$  component represents the next possibly matching clause. A choice point contains enough information for restoring the state of computation to the state before entering  $Q$ . When no more alternatives remain, the choice point is discarded.

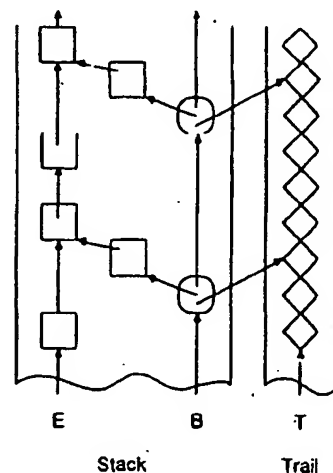


FIGURE 3. The Trail and the Tree Structure of the Stack

All choice points form a list linked by the  $B$  fields.

Note that the top of local stack is not recorded explicitly. Instead, when an environment or a choice point is established, it is placed at the next free location, computed as

$$\max(B, E + \text{env\_size}(L))$$

where  $\text{env\_size}(L)$  refers to the second operand of the call instruction (see below) pointed to by the continuation program counter, as this indicates the size of the current environment. Thus a choice point "freezes" all existing structures on the local stack, preventing them from being physically deallocated or written over.

### The Trail

This area records *conditional* variable bindings. A variable is conditionally bound if and only if the variable is older than the latest choice point. For heap variables, WAM compares the address of the variable with the H field of the latest choice point. For stack variables, the address of the variable is compared with the address of the latest choice point, that is, with B. Upon backtracking, entries are simply popped off of the trail stack and the bound variables are reset to unbound.

### Treatment of Variables

The abstract machine distinguishes two kinds of variables. A *permanent* variable is one that occurs in more than one body goal, counting the head as part of the first body goal. Thus a permanent variable has to be kept across one or more procedure calls, and so needs an environment slot. Slots are assigned in such a way that they can be discarded as early as possible: the permanent variable whose last occurrence is earliest will get the slot with the highest location. Thus environments shrink by zero or more slots for each body goal, releasing local stack space piecewise rather than an environment at a time. This optimization is called "environment trimming" and generalizes tail recursion optimization.

A *temporary* variable is any variable not classified as permanent. Temporary variables temporarily hold data during unification and are used for passing arguments in procedure calls. Procedures do not need to preserve temporary variables.

Consider, for example, the clause

```
p(A,B,S0,S)
:- q(A,B,C,C), r(S0,S1), r(S1,S).
```

where A, B, and C are temporary and all other variables are permanent.

### THE INSTRUCTION SET

Prolog programs compile into sequences of instructions, approximately one instruction per Prolog symbol. Instructions can take up to two operands, identifying temporary variables (Xi), permanent variables (Yi), small integers (N), constants (C), functors (F), and procedures (P). The instruction set is classified into *get*, *put*, *unify*, *procedural*, and *indexing* instructions. We consider the following example when discussing the instruction set. For reasons of space, formal definitions of the instructions cannot be included.

1. concatenate([ ], L, L).
2. concatenate([X|L1], L2, [X|L3])  
:- concatenate(L1, L2, L3).

A naive assignment of temporary variables in clause 2 is to reserve X1-X3 for the arguments and X4-X7 for X, L1, L2, and L3, respectively. There are no permanent variables.

### Get Instructions

Get instructions correspond to head arguments. They match against the procedure's arguments, passed in argument registers:

```
get_variable Vn,Ai
    % Vn is assigned the value of Ai
get_value Vn,Ai
    % The values of Vn and Ai are unified
get_constant C,Ai
    % The value of Ai is unified with the constant C
get_nil Ai
    % The value of Ai is unified with the constant []
get_structure F,Ai
    % The value of Ai is unified with the structure F(..)
get_list Ai
    % The value of Ai is unified with a list
```

Here, Ai denotes an argument register, which is just another name for a temporary variable. The abbreviation Vn denotes a permanent or temporary variable. The suffix (constant, list, etc.) suggests what the argument should match. The variable suffix stands for the first occurrence of a variable; value is used for subsequent occurrences.

The head arguments of clause 2 compile to

```
get_list A1          % concatenate ([
unify_variable X4     %                X|
unify_variable X5     %                L1]),
get_variable X6,A2    %                L2,
get_list A3           %                [
unify_value X4         %                X|
unify_variable X7     %                L3])
```

### Put Instructions

Put instructions correspond to body arguments. They load arguments into argument registers.

```
put_variable Vn,Ai
    % Vn and Ai are assigned a new variable
put_value Vn,Ai
    % Ai is assigned the value of Vn
put_constant C,Ai
    % Ai is assigned the constant C
put_nil Ai
    % Ai is assigned the constant []
put_structure F,Ai
    % Ai is assigned the structure F(..)
put_list Ai
    % Ai is assigned a list
```

The arguments of the goal of clause 2 compile to

```
put_value X5,A1 % concatenate (L1,
put_value X6,A2 %                L2,
put_value X7,A3 %                L3)
```

### Unify Instructions

Unify instructions correspond to arguments of a list or structure and operate in *read mode* or *write mode*, as indicated by the RW bit. If  $A_i$  contains a structure with functor  $F/n$ , the instruction `get_structure F, n,  $A_i$`  will put WAM in read mode and set the S register pointing at the arguments. Following the `get` instruction,  $n$  *unify* instructions will match subterms accessed via the S register, running in read mode. If, instead,  $A_i$  contains an uninstantiated variable, the `get` instruction will bind this variable to a structure that is about to be created, place  $F/n$  at the top of the heap, and put WAM in write mode. The *unify* instructions will fill in the subterms, running in write mode.

The `get_list` instruction operates similarly. The `put_structure` and `put_list` instructions always put WAM in write mode.

#### read mode semantics

`unify_variable Vn`  
 $\% Vn$  is assigned the next subterm  
`unify_value Vn`  
 $\%$  The value of  $Vn$  is unified with the next subterm  
`unify_constant C`  
 $\%$  The next subterm is unified with the constant  $C$   
`unify_nil`  
 $\%$  The next subterm is unified with the constant  $()$

#### write mode semantics

`unify_variable Vn`  
 $\%$  A new variable is stored in  $Vn$  and as the next subterm  
`unify_value Vn`  
 $\%$  The value of  $Vn$  is stored as the next subterm  
`unify_constant C`  
 $\%$  The constant  $C$  is stored as the next subterm  
`unify_nil`  
 $\%$  The constant  $()$  is stored as the next subterm

Note that there are no *unify* instructions for lists or structures, as they would require extensions to the WAM data areas or registers. Instead, structures are "flattened" by introducing temporary variables. For instance, a head

$p([d(0, a)])$

could compile to

```
get_list Ai          % p ( (
unify_variable Xj    %      T1
unify_nil            %      ) ) :-
get_structure d/2, Xj % T1 = d(
unify_constant 0     %      0,
unify_constant a     %      a)
```

As a goal, it could compile to

```
put_structure d/2, Xj % T1 = d(
unify_constant 0     %      0,
unify_constant a     %      a),
put_list Ai          % p ( (
unify_value Xj       %      T1
unify_nil            %      ) )
```

### Procedural Instructions

These instructions correspond to the head and goals of a clause. They deal with control transfer and environment handling:

`proceed`  
 $\%$  branch to the continuation program pointer  
`execute Q`  
 $\%$  branch to the procedure  $Q$   
`call Q, N`  
 $\%$  set the continuation program pointer at the next instruction,  
 $\%$  then branch to the procedure  $Q$   
`allocate`  
 $\%$  establish an environment on the stack  
`deallocate`  
 $\%$  logically discard an environment

Clauses with zero, one, or more goals are translated according to the pattern:

$F.$	$F :- G.$	$F :- G, H, K.$
<code>get args of F</code>	<code>get args of F</code>	<code>allocate</code>
<code>proceed</code>	<code>put args of G</code>	<code>get args of F</code>
	<code>execute G</code>	<code>put args of G</code>
		<code>call G, N</code>
		<code>put args of H</code>
		<code>call H, N1</code>
		<code>put args of K</code>
		<code>deallocate</code>
		<code>execute K</code>

where  $N \geq N1$  indicate the size of the part of the current environment which is active after the `call` instruction. Thus  $N$  permanent variables are live after the first `call`;  $N1$  are live after the second `call`. It is this operand which is referred to as `env_size(L)` in the description of the local stack above.

### Indexing Instructions

For a given call, these instructions filter out the set of possibly matching clauses of a procedure. This function is based on the principal functor of the first argument. It is guaranteed that all possible matches can be eventually tried by backtracking. There are two kinds of indexing instructions: instructions that discriminate on the first argument,

`switch_on_term Lvar, Lconst, Llist, Lstruct`  
 $\%$  dispatch on the type of the first argument

```

switch_on_constant N, Table
    % dispatch on the value of the first argument,
    or fail if not found
switch_on_structure N, Table
    % dispatch on the principal functor of the first
    argument, or fail

and instructions that backtrack over alternatives,

try_me_else L
    % a new alternative program point is L
retry_me_else L
    % set alternative program point to L
trust_me_else_fail
    % discard latest alternative program point

try L
    % the next instruction is a new alternative program
    point, goto L
retry L
    % the next instruction replaces alternative program
    point, goto L
trust L
    % discard latest alternative program point, goto L

```

The clauses of `concatenate/3` are linked together as follows:

```

switch_on_term $1, $2, $4, fail
$1: try_me_else $3
$2: /* the code for clause #1 */
$3: trust_me_else_fail
$4: /* the code for clause #2 */

```

The first instruction does a four-way dispatch on the type of the first argument: if it is a variable, both clauses are possible and so the `try_me_else` instruction establishes a choice point whose alternative label is `$3` before executing clause 1. If WAM backtracks, the `trust_me_else_fail` instruction will erase the choice point. If the first argument is, respectively, a constant or a list, then the only possible alternatives are clause 1 or 2 (label `$2` or `$4`), respectively. If it is a structure, the call fails.

The `retry_me_else` instruction is used for alternatives except the first and last ones.

The other `switch` instructions are used when there are several clauses with a constant or a structure as first head argument, respectively. These instructions provide further filtering of the set of possible matches by doing a hash table lookup with the principal functor as key.

The `try/retry/trust` instructions are used when there are several clauses with a first head argument with the same principal functor.

### Compiler Optimizations

There are several opportunities for compiler optimizations. For instance, the two instructions

```

get_variable Xj, Aj
put_value Xj, Aj

```

represent noops and can be deleted, and a compiler could try to minimize the size of the emitted code by properly allocating temporary variables. Thus an optimized version of clause 2 above is

```

get_list A1
    % concatenate ( (
unify_variable X4
    %
    % X1
unify_variable A1
    %
    % L1), L2,
get_list A3
    %
    % (
unify_value X4
    %
    % X1
unify_variable A2
    %
    % L3)) :-
execute concatenate/3
    % concatenate (L1, L2, L3).

```

This version is four instructions shorter than the naive translation given earlier.

### BASIC GARBAGE COLLECTION ALGORITHM

We first describe a rudimentary garbage collection algorithm, and in the subsequent sections we enhance it to take more advantage of the specific properties of Prolog. Our garbage collection algorithm consists of marking and compaction.

#### The Marking Phase

During this phase all reachable objects from a set of roots are marked by setting the *m*-bits of the value cells. The *f*-bit is initially FALSE but is temporarily used and reset to FALSE at the end of the marking phase.

*Prerequisites for the marking phase.* In the original WAM, some value cells on the stack may be uninitialized. A straightforward way to ensure that every environment is fully initialized at every procedure call requires the following changes in the WAM [6]:

- (1) All environment slots are initialized to unbound before the first call by inserting sufficient `put_variable Yn` instructions.
- (2) As a compensation we may optimize the WAM code after the first call by changing every `put_variable Yn` into `put_value Yn`.
- (3) In the body of a clause the semantics of `unify_variable Yn` are modified so that *Yn* is trailed if the computation is in a nondeterministic state. This will ensure that the variable will be reset on backtracking, thus avoiding the possibility of dangling references.

In the following we assume that garbage collection is triggered at a well-defined point, for instance, just after the `call` instruction. The arity of the predicate called indicates the number of active argument registers.

### The Marking Algorithm

```
marking_phase()
{
    mark_registers();
    /* the A registers */
    mark_environments(E, Env_size(L));
    /* active environments */
    mark_choicepoints(B);
    /* choice points and environments
       reachable from choice points */
}
```

First all structures on the heap accessible from the active argument registers are marked. Since most computer architectures cannot handle a reference to a register and the `mark_variable` routine (defined later) takes a reference to a value cell, each register is first moved to the local variable `temp`. The notation `&temp` used below means the address of `temp`.

```
mark_registers()
{
    struct valuecell temp;
    n = number of active registers A;
    for i = 1 up to n
        if (A[i] points to a heap cell)
        {
            temp = A[i];
            mark_variable(&temp);
        }
}
```

Then all active environments are marked. The environment size of `E` is computed as `env_size(L)`.

In Figure 4(a), the chain of active environments reachable from the current environment `E` is marked, indicated by the shaded cells. From each choice point there is another chain of environments, as can be seen from the next figure. Some environments are only

reachable from the choice point (2) and some are shared (3, 4, 6). The first shared environment (3) is a bit special: some cells within it may only be reachable from the chain starting at the choice point. Those cells are always the last cells allocated in that environment.

As illustrated by Figure 4(c), only a part of the active environment stack needs marking when accessed from a choice point. In our marking algorithm this optimization is implemented by first checking to see whether the given stack cell has already been marked; if it has, then we immediately stop marking that chain of environments. To make this work properly two observations have to be made.

- (1) The value cells of each environment have to be traversed from high to low, that is, from the top of the stack toward the bottom. By doing so, we first encounter the possibly unmarked cells of a shared environment.
- (2) When marking a value cell on the stack, a reference within the stack is ignored. This is quite safe to do since WAM guarantees that all references within the stack are within the same chain of environments. Thus the referenced cell will be marked anyway.

The routine that marks a chain of environments is called `mark_environments`:

```
mark_environments(env, size)
struct environment*env;
int size;
{
    while (env != NULL)
    {
        for each v pointing to env → Y[size]
            down to env → Y[1]
            if (v → m == TRUE)
                return;
    }
}
```

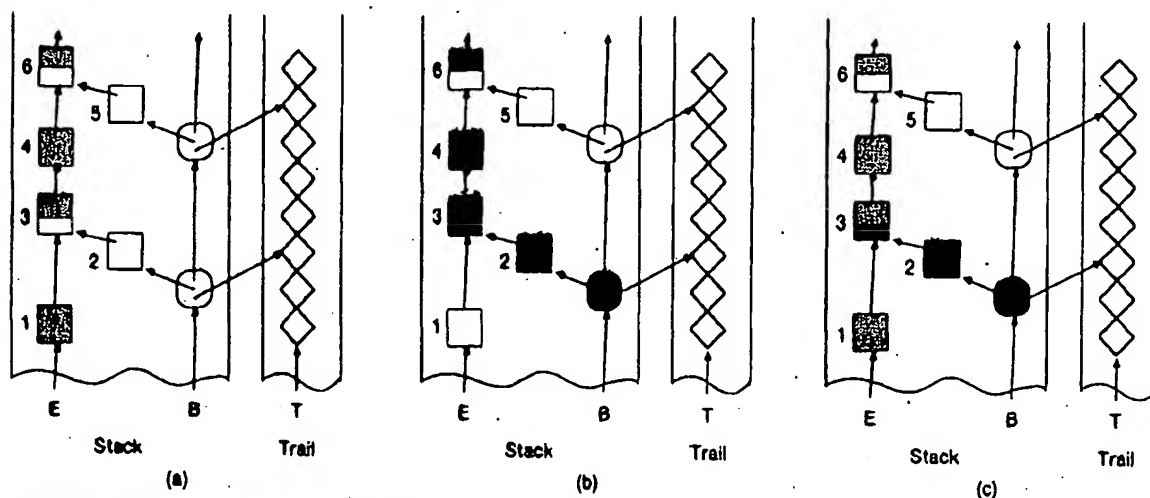


FIGURE 4. Marking the Environments. (a) Cells Reachable from Current Environment. (b) Cells Reachable from a Choice Point. (c) Part of the Stack that Actually Needs Marking from a Choice Point Shown in Black.

```

    else if (v → value points to the
    heap)
        mark_variable(v);
    size = env_size(env → CL);
    env = env → CE;
}

```

All choice points and the corresponding chains of environments are marked by `mark_choicepoints`:

```

mark_choicepoints(cp)
    struct wam *cp;
    while (cp ≠ NULL)
    {
        mark_environments(cp → E, env_size
        (cp → L));
        for each v pointing to a valuecell in
        cp → A do
            if (v → value points to the heap)
                mark_variable(v);
        cp = cp → B;
    }

```

*A note on the trail.* Any variable recorded in the trail is also accessible from some choice point. Thus all variables reachable from the trail have already been marked and the trail need not be scanned. Later, in the section, Early Reset of Variables, we elaborate further on this subject.

Pittomvils et al. [13] present a similar way to traverse the environments and the choicepoints, but give no details on how to mark the variables. Before we present the procedure `mark_variable`, some background information on the marking phase is first given. Two concepts are of importance: chains and structures. A *chain* is a linked list of value cells. The cells of a chain are classified as follows. A *head of chain* is either a cell on the stack or any cell in a structure except the first one. A *last cell of chain* is either a cell tagged as a `CONSTANT` or it is a cell previously investigated during the

marking phase. An investigated cell either has its *m*-bit or *f*-bit set to `TRUE`. Any cell in a chain different from the head of the chain is called an *internal cell*.

In WAM an unbound variable is represented by a cell pointing to itself. Such a cell forms a chain by itself, in which case the head and the last cell of the chain coincide. Figure 5 shows a chain starting from the stack where the last cell is a constant. The arcs of the chain are marked 1, 2, 3, 4.

Although cyclic structures do not normally occur in Prolog, Figure 6 shows a chain starting from the second cell of a list and ending in a cell completing a cycle. The arcs of the chain are marked 1, 2, 3. Notice that the chain consists of four cells, the cell between arc 1 and 2 is both the second and last cell of the chain.

We are now ready to present the core of the marking algorithm, the procedure `mark_variable`. The following algorithm uses a new pointer reversal algorithm especially adapted for WAM data structure. An extra stack is thus unnecessary. Also it is capable of marking cells within structures, creating situations where part of the structure is collected as garbage and part of it is not. This is important since pointers into structures are quite frequent.

The algorithm is described by a finite-state machine with two states: *forward* and *backward*. (See Figure 7.) The forward phase traverses a chain of pointers that becomes reversed. During this phase, when an unmarked structure or list is reached, a return pointer to the current chain is saved in the last cell of the structure. We then continue the forward phase with a subchain originating in this cell. Alternatively, if a constant or a marked cell is encountered, the backward phase is entered.

In the backward phase, pointer reversal is undone until we return to the starting point denoted by the head of the current chain. If this cell is the starting point of the whole marking phase, we are finished. Otherwise, the cell is a component of a structure and the traversal continues in the forward mode with the previous component in the structure.

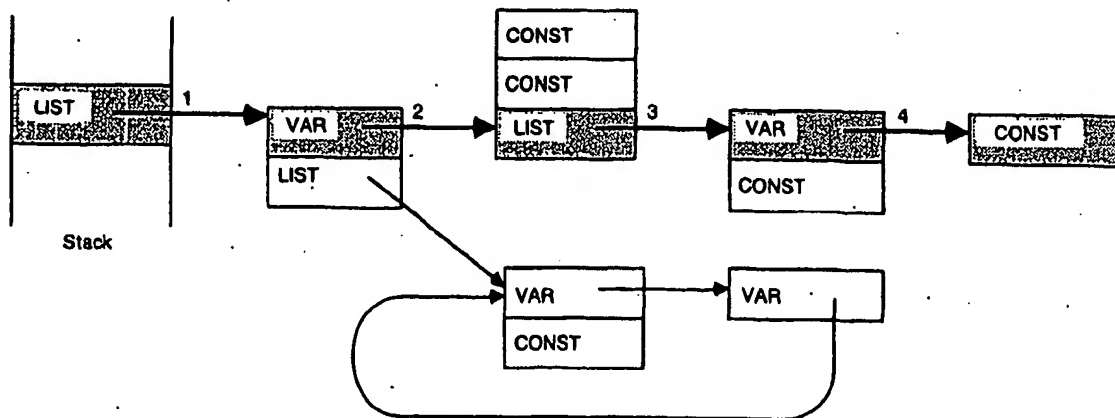


FIGURE 5. A Chain Starting from a Stack when the Last Cell is a Constant

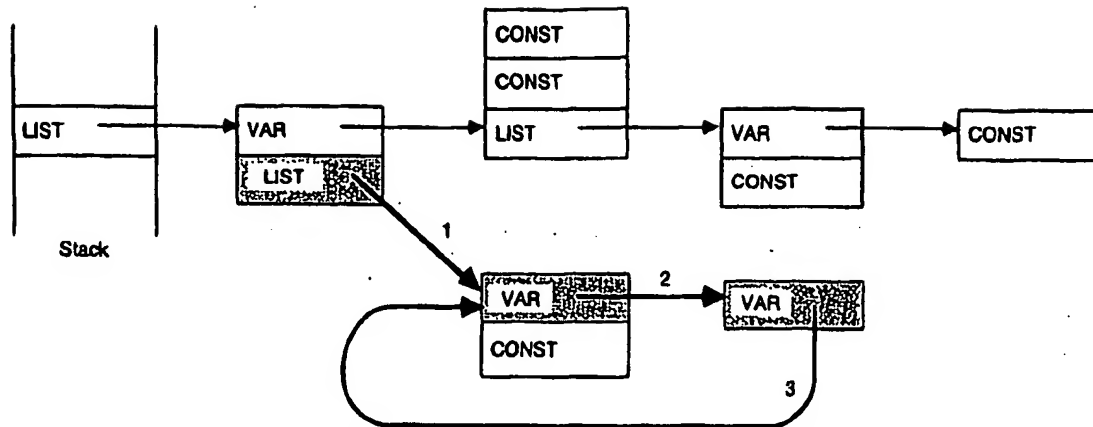


FIGURE 6. A Chain Ending in a Cell that Completes a Cycle

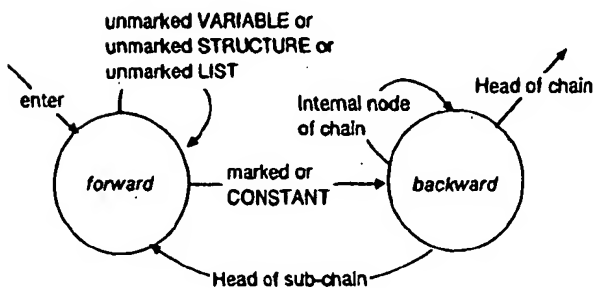


FIGURE 7. The Algorithm Described as a Two-State Machine

Two pointers for traversing the data structures are used during this phase: *current* and *next*. *Current* points to the cell currently being processed, whereas *next* contains the original value of what *current* pointed to. The following is a detailed description of the two phases of the marking algorithm.

#### The Forward Phase

Initially *current* points to a stack cell which has its f-bit set, and *next* contains the value of that cell. Execution may then enter the forward phase:

- (1) If *current* points to an unmarked VARIABLE

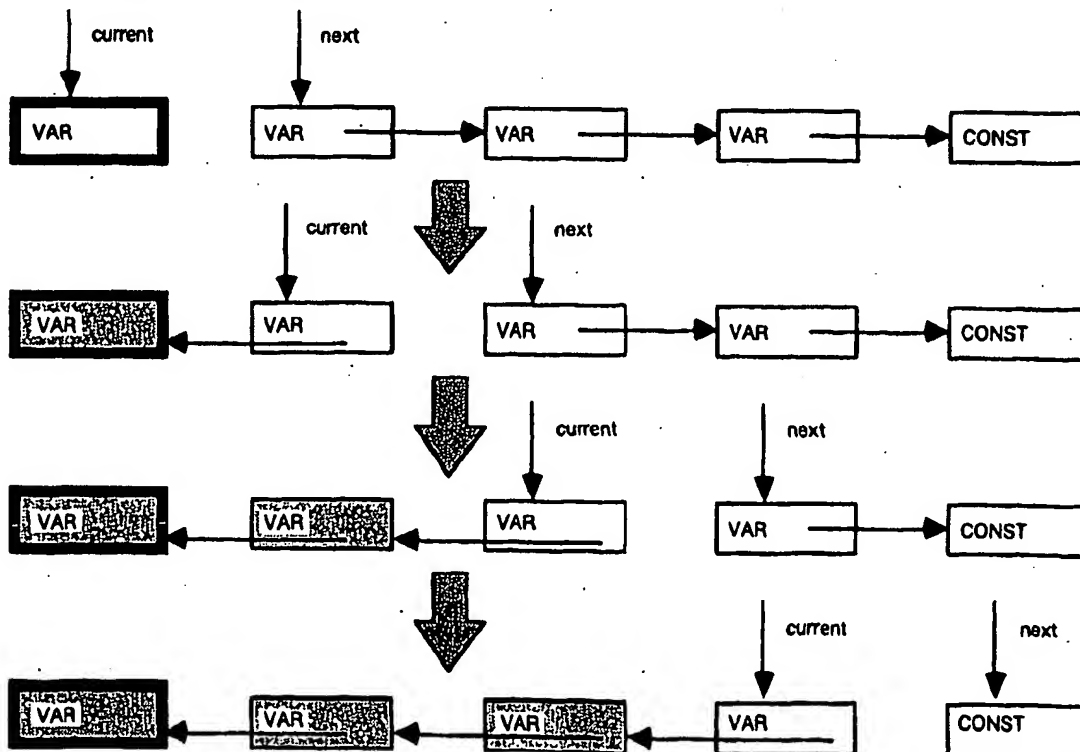


FIGURE 8. Transformation 1

cell, we get the state transition depicted by the following figure where shaded cells denote cells having their m-bit set and bold outlined cells denote cells having their f-bit set. The f-bit is used to distinguish the "head of chain" from an internal cell. Note that the new current cell becomes marked as an internal cell. After the transition, execution continues in the forward mode. Figure 8 shows several such transitions.

(2) If *current* points to an unmarked **STRUCTURE** or an unmarked **LIST**, a fairly complicated transition takes place, as can be seen from Figure (9a). First the current cell is marked and all the components of the structure next points to are examined. If the second component has its f-bit set, the structure is already being traversed, and execution continues in the backward phase. Otherwise, the f-bit is set in all compo-

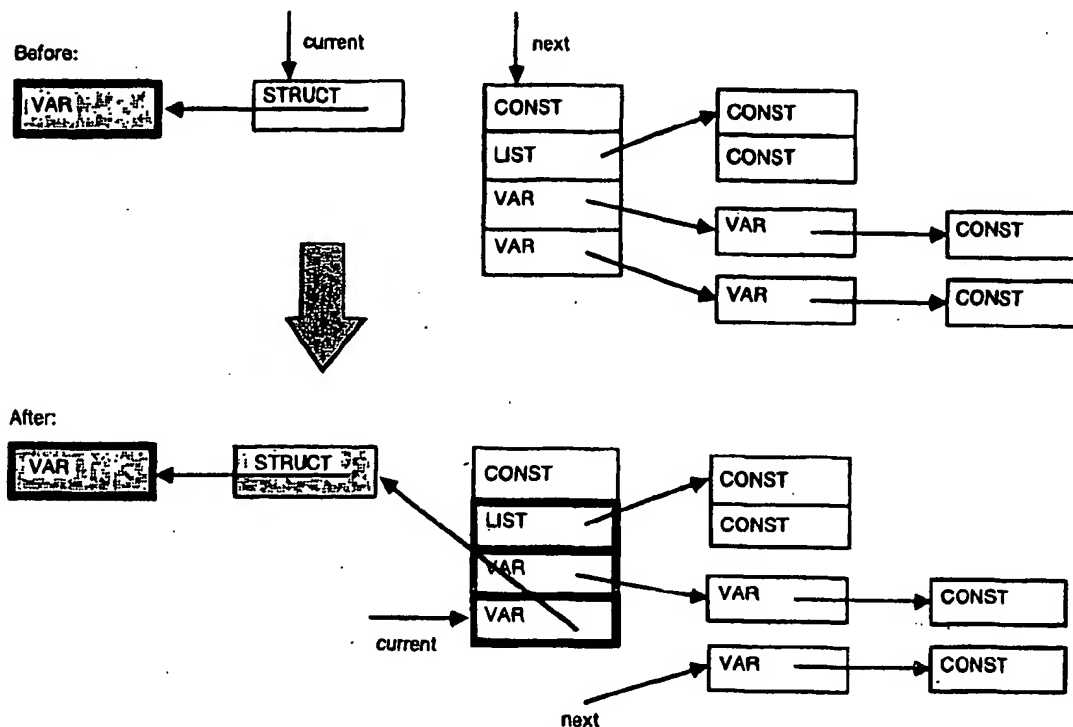


FIGURE 9. (a) Transformation 2a

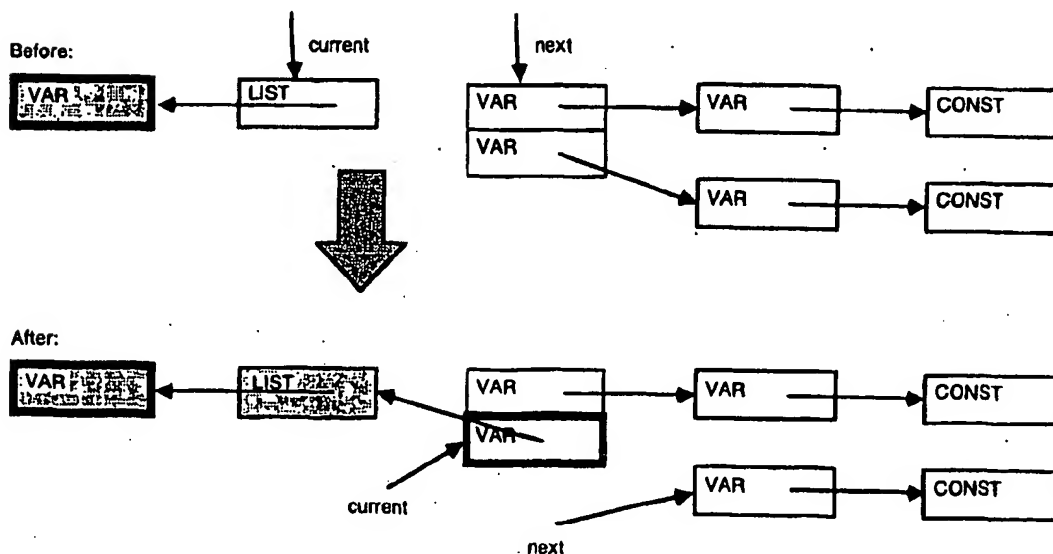


FIGURE 9. (b) Transformation 2b

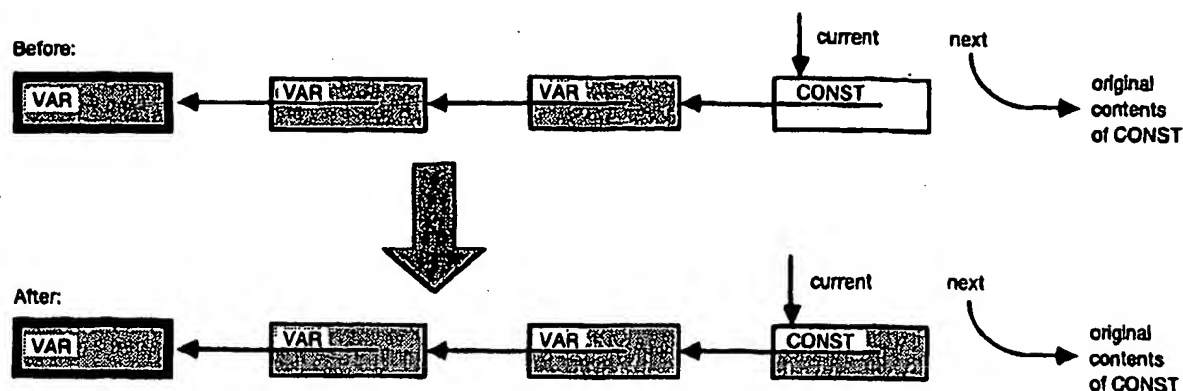


FIGURE 10. Transformation 3

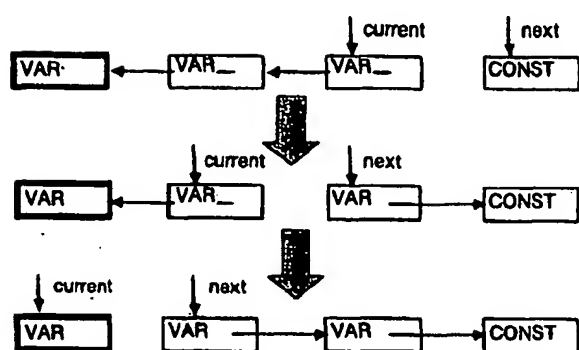


FIGURE 11. Transformation 4

nents but the first. This indicates that all these components are heads of subchains. Thereafter the pointers are arranged so that execution may continue in the forward mode with *current* pointing to the last cell of the structure. Figure 9(b) shows an example where *current* points to a list cell.

(3) If *current* points to a marked cell or an unmarked CONSTANT cell, as shown in Figure 10, the current cell becomes marked if unmarked, and the backward phase is entered.

#### The Backward Phase

This phase resets the pointers of a chain to their original values. (See Figure 11.) All internal cells are reversed until the "head of chain," which has its f-bit set, is encountered. Since the f-bit now has fulfilled its pur-

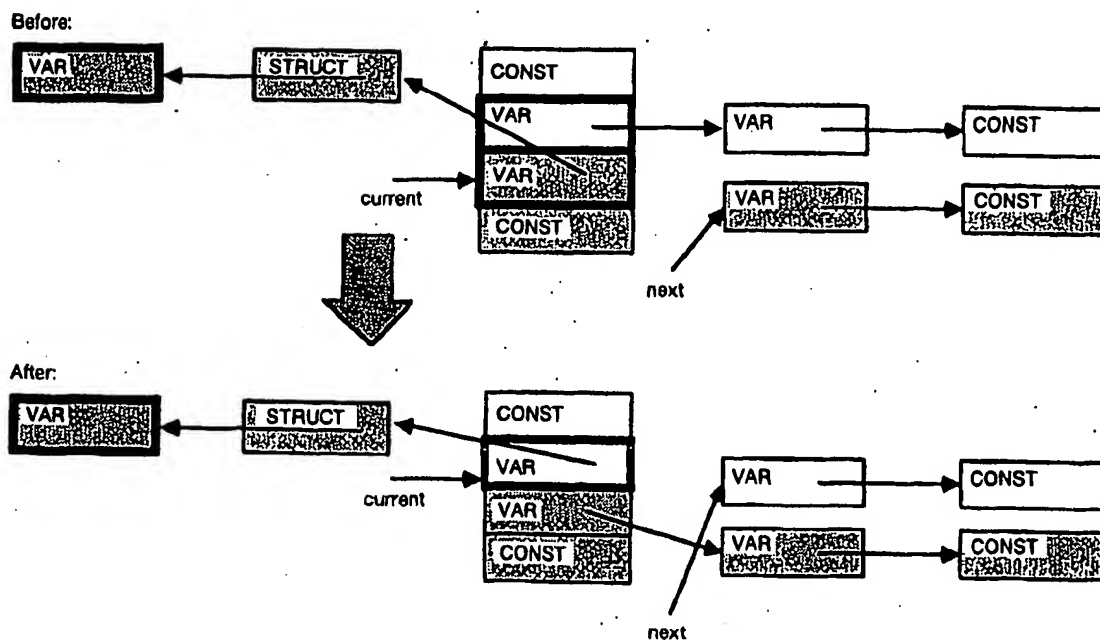


FIGURE 12. Transformation 5

pose (being the indicator for the "head of chain"), it is reset. We have then either arrived at the starting cell of the call to `mark_variable` and the marking terminates, or it is a cell not being the first in a LIST or a STRUCTURE. In the latter case, as the cells of a structure are investigated from the last to the first, `current` is advanced to point to the previous cell. The pointers are then arranged so that execution may continue in the forward mode; see Figure 12.

Before we are ready to show the code for the `mark_variable` procedure, some help-macros need to be defined. To simplify the definitions of those help-macros, a help-help-macro is first defined:

- (1) `Swap3(x, y, z)` means `temp = x;`  
`x = y; y = z; z = temp;`

The help-macros then get the following definitions:

- (2) `Reverse(current, next)` means `Swap3`  
`(next → value, current, next).`  
 (3) `Undo(current, next)` means `Swap3` `(current → value, next, current).`  
 (4) `Advance(current, next)` means `Swap3`  
`((current + 1) → value, next, current`  
`→ value).`  
 (5) `Lastcell(current, next)` returns a pointer to the last component of the structure `current` points to. It may be necessary to use `next` for this calculation as it contains the original value of `current → value`, which contained the length of the structure. For a list, however, the address of the last component is always `next + 1`.

During the marking, the number of marked cells is calculated by incrementing the variable `total_marked`. In order not to count the cells on the stack, `total_marked` is decremented at the start.

```
mark_variable(start)
  struct valuecell *start;
  {
    struct valuecell *current, *next;
    current = start;
    next = current → value;
    current → f = TRUE;
    total_marked = total_marked - 1;
    /* don't count stack cells */
    goto forward

  forward: if (current → m == TRUE) goto
  backward;
    current → m = TRUE;
    total_marked = total_marked + 1;
    switch (current → tag) {
    case VARIABLE:
      /* transformation 1 */
      if (next → f == TRUE)
        goto backward;
      Reverse(current, next);
      goto forward;
    case STRUCTURE:
```

```
      /* transformation 2a */
    case LIST:
      /* transformation 2b */
      if ((next + 1) → f == TRUE)
        goto backward;
      /* setting the f-bits */
      for every cell in object referred
        by next,
        except the first component
          set f bit TRUE;
      next = Lastcell(current, next);
      Reverse(current, next);
      goto forward;
    case CONSTANT:
      /* transformation 3 */
      goto backward;
    }
  }

backward: if (current → f == FALSE)
  /* transformation 4 */
  { /* internal cell */
    Undo(current, next);
    goto backward;
  }
  /* head of chain */
  current → f = FALSE;
  if (current == start)
    return;
  current = current - 1;
  /* transformation 5 */
  Advance(current, next);
  goto forward;
}
```

#### Optimizations of `mark_variable`

Each time an object is marked, all its components are individually investigated, regardless of whether the object has been marked before or not. It is wrong to assume that the whole object has been marked if a single component has been marked, since individual components may be marked if they are referred from VAR-cells. Investigating all the components may be time consuming, although the algorithm is still linear in principle.

In order to speed up execution, we may check all of the cells to see whether they are marked. The following code, inserted just before setting the f-bits of the components, will accomplish this:

```
if (all cells in object referred by
    next have their m bit TRUE)
  goto backward;
```

An even better solution is to use the observation that the first component of a structure may never be referred to directly from a VAR-cell, and thus may never be marked individually. Since the first cell is the last cell marked, it is sufficient to test this cell alone to know whether the structure as a whole has been marked. We then get the following piece of code which is inserted just before the setting of the f-bits:

```

if (current → tag == STRUCTURE AND
    next → m == TRUE)
    goto backward;

```

A corresponding optimization for LISTS is not possible as the first component may be reached directly from a VAR-cell.

#### Some Extreme Cases

It is not immediately obvious that the marking algorithm is able to handle quite complex structures. We have already mentioned the ability of the algorithm to mark only those *parts* of a structure that are actually referred to. If the rest of the structure remains unmarked, those cells will be reclaimed later. (See Figure 13.) A variant of this occurs when the first cell of a list is also referred to from a VAR (see Figure 14).

from there leads to b and then back to c, which is still unmarked, although it is part of a structure being investigated. The marking continues with e and finally returns to a, which is marked so the backward mode is entered.

On the other hand, if marking instead starts with b, passing through c, e, a and back to c, a cycle will not be formed until c. Although c is marked, we do not enter the backward mode here since d has not yet been marked. A trace of the execution would show that the marking algorithm is able to handle both these cases.

A very special cyclic structure is an unbound variable (see Figure 16). Usually, it is represented by a cell pointing to itself. The marking algorithm needs no special code for this case, although it might speed up execution.

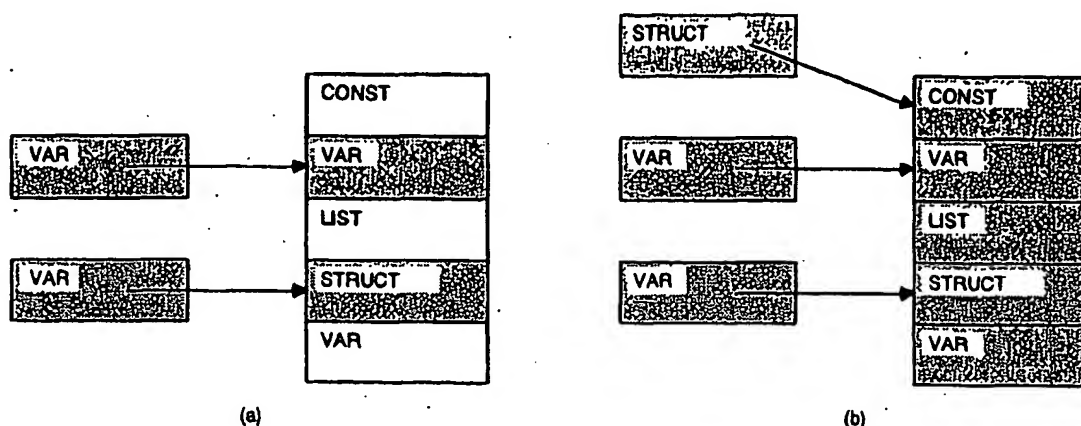


FIGURE 13. How the Marking Algorithm Handles Structures. (a) Only the Parts of a Structure Actually Referred are Marked. (b) All Components are Marked if there is a STRUCT Pointer.

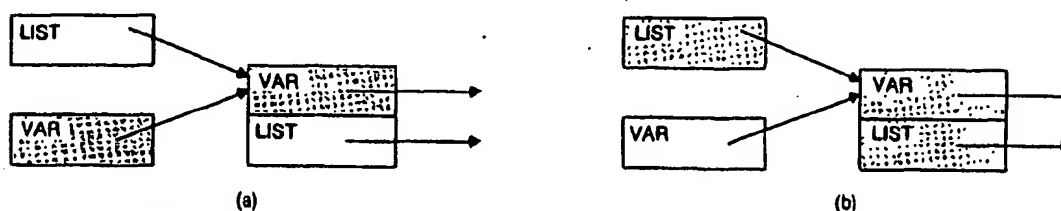


FIGURE 14. How the Marking Algorithm Handles Lists. (a) A VAR Pointer to the First Cell only Marks that Cell. (b) A LIST Pointer to the First Cell Marks Both Cells.

Although cyclic structures are seldom constructed in ordinary Prolog programs, they do occur in some applications. The marking algorithm handles them gracefully, without treating them separately. When a marked cell is encountered, the marking algorithm enters the backward mode.

Extra care was taken when constructing the algorithm so that it would correctly handle cyclic structures like the one shown in Figure 15. If the marking starts with cell a, it will first come to the two cells c and d. Since a composite object is always scanned backward, cell d will be investigated first. The reference

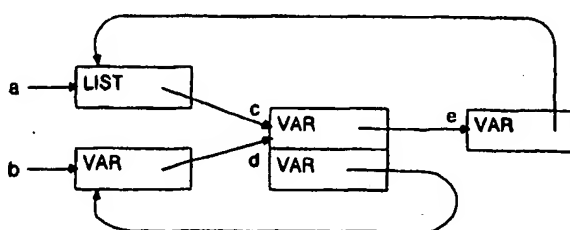


FIGURE 15. A Cyclic Structure

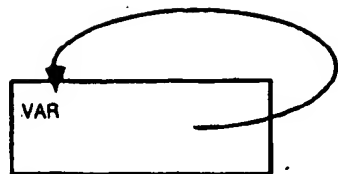


FIGURE 16. An Unbound Variable, a Special Cyclic Structure

### Informal Correctness Proof of the Marking Algorithm

It is guaranteed that the marking algorithm marks all reachable objects, resets all pointers to their original values, and terminates. The main complexity of the algorithm lies in the procedure `mark_variables`. No correctness proof is given here, but a sketch looks like this:

The following preconditions for the forward and the backward phase are always maintained:

- (1) `next` always contains the original value of `current`—value.
- (2) `current` always points to a return-chain, where a return-chain is defined as follows: `current` is the end of a chain whose first cell has its f-bit set. This first cell is either the starting cell (`start`) or a component of a structure. In the latter case, that component refers back to the `STRUCTURE` or `LIST` value cell which originally pointed to the structure. Now that `STRUCTURE` or `LIST` cell is instead part of a return chain. All components with higher addresses are marked subtrees, as shown by Figure 17. All lower components of the structure, besides the first, have their f-bits set.

For the backward phase, the following is also always true.

- (3) `next` points to a marked subtree.

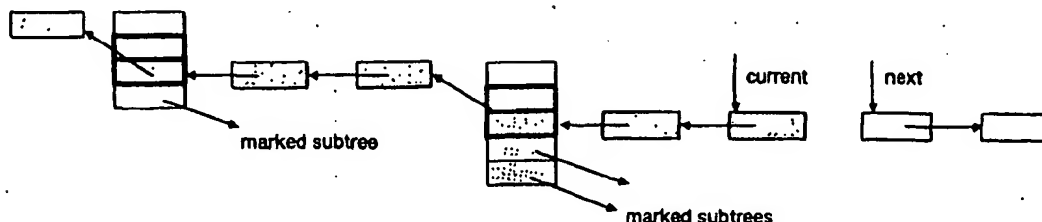


FIGURE 17. A Typical Return-Chain

The preconditions are easily shown to be true initially. By performing a case analysis, it can be shown that they are maintained throughout the computation. If the computation terminates, it will end in the backward mode, with `current` being the root of the marking and `next`, that is, `current`—value, pointing to a marked subtree.

It remains to show that the computation actually terminates. Once an object is encountered, it is always marked. If a marked object is found, the marking enters backward phase. As no two return-chains ever cross each other (thereby creating a loop) and there is only a finite number of cells, the computation must terminate.

### THE COMPACTION PHASE

Our goal for the compaction phase was to find a linear algorithm that used no extra space. As in Pittomvils [13], our algorithm is based on Morris' algorithm [9], adapted for the Prolog environment. The algorithm is of order  $n$ , requiring two passes through the heap, one pass through the trail, and one pass through the stack. The heap is scanned, once to update upward pointers and once to update downward pointers. First the basic sweeping algorithm is described and then we make some additions to make it suitable for WAM.

#### Compacting the Heap

By sweeping the heap from low to high addresses, having a destination pointer which is incremented for each marked object encountered, it is possible to know the final location of each object. Similarly, since we know how many value cells are marked, as calculated in `total_marked`, we can alternatively sweep the heap from high to low address. The main problem with the compaction phase is not, however, to calculate the final location of a certain cell, but to find and update all those cells pointing to this cell. A very natural solution would be to link all those value cells into a relocation chain, which can be scanned when the final destination of the cell is known.

Figure 18(a) shows all variable cells pointing to the cell `current`. In Figure 18(b), all those cells are linked into a chain starting at `current` and ending with a cell containing the original value of `current`. A moment of afterthought indicates, however, that this is not a possible solution. Many cells contain pointers and thus need to be part of a relocation chain; however, at the same

time they may be pointed to and thus need to be the head of another relocation chain!

Morris' algorithm elegantly solves this apparent contradiction by allowing some value cells to be the head of relocation chains part of the time and to be members of relocation chains part of the time. As mentioned previously, compaction consists of two phases, one up-

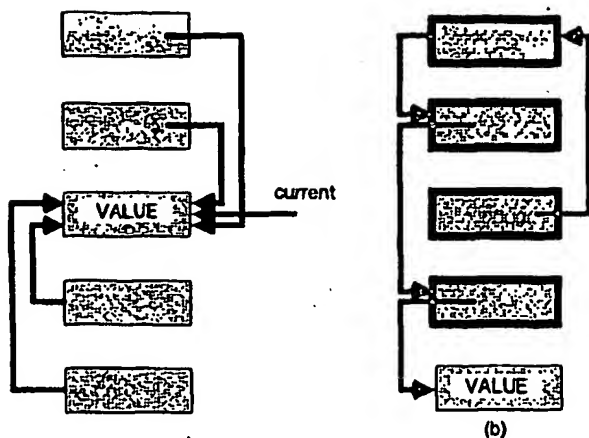


FIGURE 18. An Erroneous Attempt to Create a Relocation Chain. (a) Original Situation. (b) A Not Possible Relocation Chain for Current.

also regains its original VALUE; see Figure 20.

The value of *current* is now inspected to see whether it is an upward pointer. If so, the cell is linked into the relocation chain of the object pointed to. This is done by calling `into_relocation_chain(current - value, current)`.

Thereafter, *current* is advanced to point to the next marked value cell. In this simple example the remaining value cells all point downward and nothing more will happen during the upward phase. After the upward phase, all value cells pointing upward have their final correct values.

The downward phase is almost a mirror image of the upward phase, scanning the heap in the opposite direction. The difference is that all marked value cells, regardless of their contents, are moved to their new location and their *m* and *f* bits are reset. There is no danger that moving a cell will destroy any other cell, as the destination is always closer to the bottom of the heap

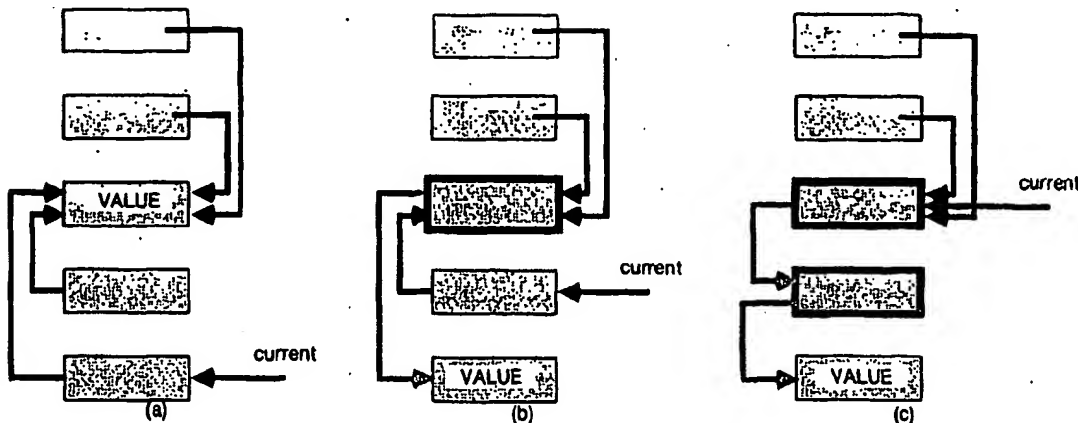


FIGURE 19. A Successful Attempt to Create a Relocation Chain. (a) Original Situation. (b) The First Valuecell is Linked into the Relocation Chain. (c) The Next Valuecell is Linked into the Relocation Chain.

ward sweep and one downward sweep. In the upward phase, the heap is scanned from high to low, and only marked value cells pointing upward are considered. Whenever such a cell is encountered, it is linked into the relocation chain of the cell it points to. The *f*-bit is used to indicate that a cell is in a relocation chain. As before, such cells are shown with a bold outline in our figures.

Each marked cell, regardless of its pointer value, is first checked to see whether it is the head of such a chain. If so, all members of the chain are updated to point to the new location of the cell. In this process, the cell also regains its original value. If the original value points upward, it must also be inserted into a relocation chain. Figure 19 shows a series of situations during the upward phase as the pointer *current* passes through the heap. *current* is now the head of a relocation chain whose members will be updated to contain the future location of *current*. In this process *current*

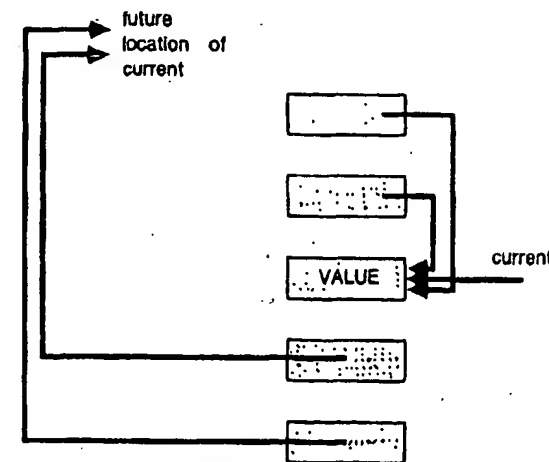


FIGURE 20. Final Processing of a Relocation Chain. Its Members are Assigned Their Future Values.

than the source and all lower cells have already been moved. The downward phase resets all downward pointers. Thus, the downward phase in combination with the upward phase assures that all pointers are correctly reset.

```
compact_heap()
{
    struct valuecell *dest, *current;
    /* the upward phase */
    dest = heap_low + total_marked - 1;
    for current = H down to heap_low
    {
        if (current → m == TRUE)
        {
            update_relocation_chain(current,
                                   dest);
            if (current → value is a heap
                pointer)
            {
                if (current → value < current)
                    into_relocation_chain(current →
                                           value, current);
                else if (current == current →
                        value)
                {
                    /* a cell pointing to itself */
                    current → value = dest;
                }
            }
            dest = dest - 1;
        }
    }
    /* the downward phase */
    dest = heap_low;
    for current = heap_low up to H
    {
        if (current → m == TRUE)
        {
            update_relocation_chain(current,
                                   dest);
            if (current → value is a heap
                pointer AND
                current → value > current)
            {
                /* move the current cell and in-
                 * sert it into the relocation
                 * chain */
                into_relocation_chain(current →
                                       value, dest);
                dest → tag = current → tag;
            }
            else
            {
                /* just move the current cell */
                dest → value = current → value;
                dest → tag = current → tag;
                dest → f = FALSE;
            }
            dest → m = FALSE;
            dest = dest + 1;
        }
    }
}
```

```

    }
}

update_relocation_chain(current, dest)
struct valuecell *current, *dest;
{
    struct valuecell *j;
    while (current → if == TRUE)
    {
        j = current → value;
        current → value = j → value;
        current → f = j → f;
        j → value = dest;
        j → f = FALSE;
    }
}

into_relocation_chain(j, current)
struct valuecell *j, *current;
{
    current → value = j → value;
    current → f = j → f;
    j → value = current;
    j → f = TRUE;
}

```

#### Updating Pointers from Other Data Areas Into the Heap

As mentioned initially, the heap is not the only data area containing heap pointers. The stack, the trail, and the argument registers A also refer to the heap, and those pointers also have to be updated. By simply sweeping those areas looking for pointers into the heap and inserting those cells into the relocation chain, the compact\_heap procedure will automatically update those references.

The main procedure for the compaction then becomes

```
compaction_phase()
{
    push_registers();
    sweep_trail();
    sweep_stack();
    compact_heap();
    pop_registers();
}
```

#### Sweeping the Registers

The registers are temporarily pushed onto the trail so that they may be referred to from the heap. The updated registers are finally by pop\_registers.

```
push_registers()
{
    n = number of active registers in A;
    for i = 1 up to n
        push_on_trail(A[i]);
}

pop_registers()
```

```

{
  n = number of active registers in A;
  for i = n down to 1
    A[i] = pop_from_trail();
}

```

### Sweeping the Trail

The procedure `sweep_trail` inserts the cells of the trail into the relocation chains. Note that the registers being pushed onto the trail will be handled here also.

```

sweep_trail()
{
  struct valuecell *current;
  for current = T down to trail_low
    if ((*current) is a heap pointer)
      into_relocation_chain(*current,
                           current);
}

```

### Sweeping the Stack

`sweep_stack` calls both `sweep_environments` and `sweep_choicepoints` in a manner very similar to the marking phase. One difference is that the *m* bits are reset during the traversal, so that upon completion all *m* bits are reset in the stack. If an unmarked variable is encountered in `sweep_environments`, the procedure returns immediately since that environment must already have been encountered.

Extra care has to be taken with the saved top-of-heap pointers in the choice points, that is, the *H* component of a choice point. This pointer indicates how far to reset the top of heap on backtracking. Since the objects in the heap may move, it may also be necessary to update this pointer. It is not sufficient just to insert the *H* component into the relocation chain of the value cell referred because that cell may be unmarked. A simple and effective solution to this problem is simply to mark that cell and insert something harmless there, for instance, `NIL`.

Alternatively, the heap may be searched until a

marked object is found and then the *H* component changed to point to that object instead. The advantage of the former solution is that no search is needed, whereas the latter avoids allocating an extra cell. (See Figure 21.)

The best solution is probably a compromise, where the heap is searched for a limited number of cells, and only if no marked cell is found is an unmarked cell then marked.

For simplicity, the code presented here uses the first alternative.

```

sweep_stack()
{
  sweep_environments(E, env_size (L));
  sweep_choicepoints(B);
}

sweep_environments(env, size)
  struct environment *env;
  int size;
{
  while (env ≠ NULL)
  {
    for each v pointing to a valuecell
      in env — Y
        scan the 'size' number of cells
          from high to low
            if (v — value points to the
              heap)
              {
                if (v — m == FALSE)
                  /* we have already been here */
                  return;
                else
                {
                  v — m = .FALSE;
                  into_relocation_chain
                    (v — value, v);
                }
              }
  }
}

```

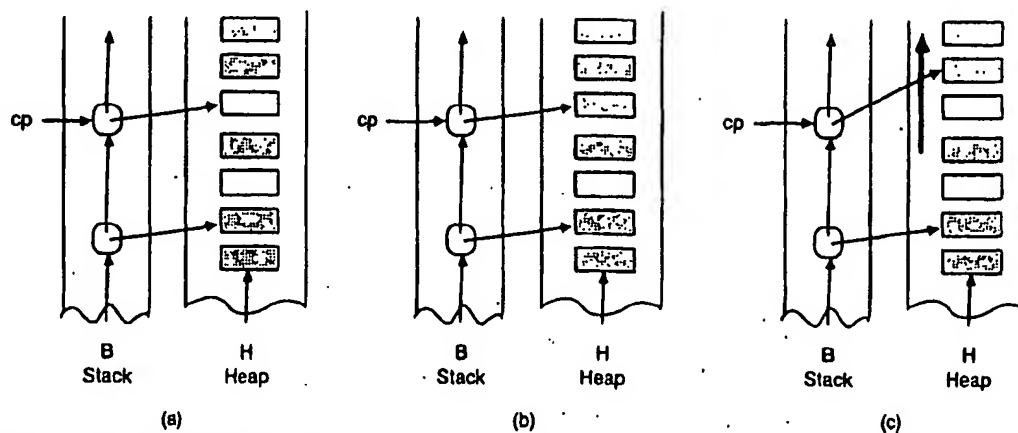


FIGURE 21. Handling References to Unmarked Cells in the Heap. (a) Problem:  $CP \rightarrow B.H$  Points to an Unmarked Cell. (b) Solution 1: Mark the Cell. (c) Solution 2: Find a Marked Cell.

```

    size = env_size(env → CL);
    env = env → CE;
  }

sweep_choicepoints(cp)
  struct wam 'cp;
  {
    while (cp ≠ NULL)
    {
      sweep_environments(cp → E, env_size
        (cp → L));
      for each v pointing to a valuecell
        in cp → A do
        if (v → value points to the heap)
        {
          v → m = FALSE;
          into_relocation_chain
            (v → value, v);
        }
      if (cp → H → m == FALSE)
      {
        /* create a dummy constant on the
           heap */
        cp → H → value = NIL;
        cp → H → tag = CONSTANT;
        cp → H → m = TRUE;
        cp → H → f = FALSE;
        total_marked = total_marked + 1;
      }
    }
  }

```

```

    into_relocation_chain
      (cp → H, &(cp → H));
    cp = cp → B;
  }

```

As already mentioned, the notation  $\&(cp \rightarrow H)$  means the address of  $(cp \rightarrow H)$ .

#### Putting the Pieces Together

Having defined the marking and compaction phases, the basic garbage collection algorithm now becomes simply

```

garbage_collection()
{
  marking_phase();
  compaction_phase();
}

```

This concludes the basic garbage collection algorithm.

#### OPTIMIZATIONS OF GARBAGE COLLECTION FOR WAM

##### Early Reset of Variables

As shown by the following Prolog program, some data structures will be created which are not useful, but which will be marked anyway by the basic garbage collection algorithm.

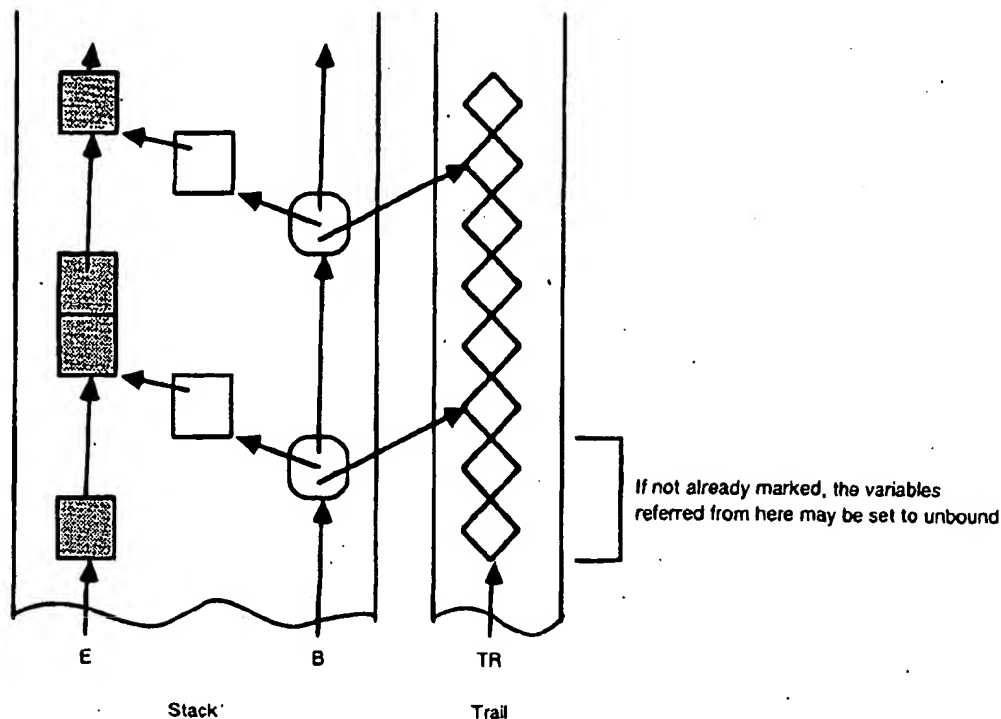


FIGURE 22. If Not Already, Computation State Where Early Reset May Be Applicable

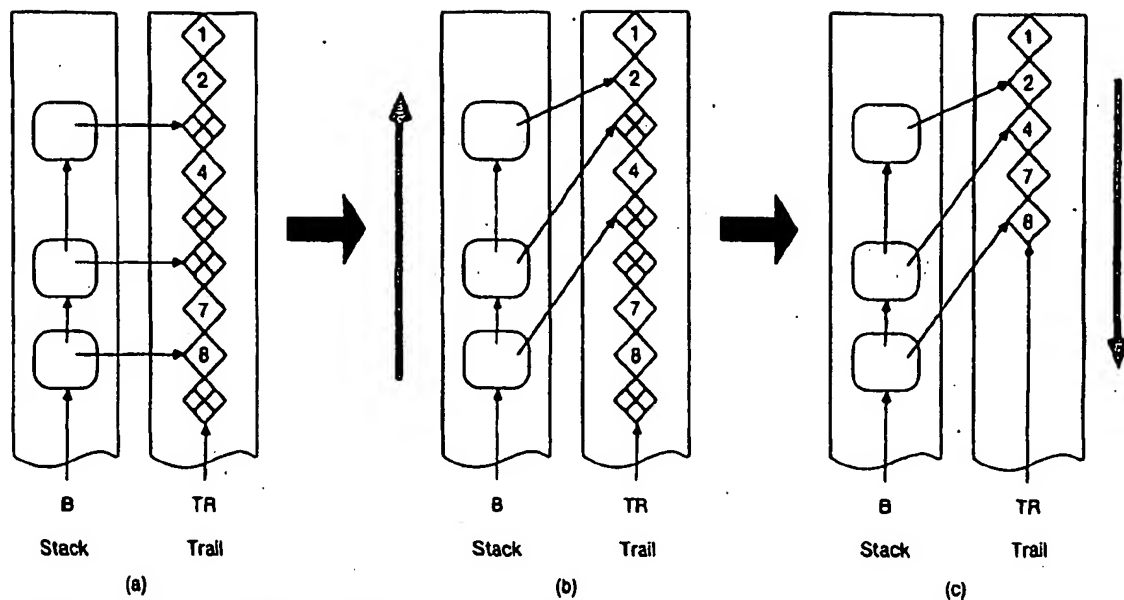


FIGURE 23. Sliding the Trail. (a) Initial State. Trail Cells Containing NULL Are Marked X. (b) State After Updating the Choice Points. (c) State After Sliding the Trail.

```
test(B)
:- p(A, B).
p(A, B)
:- create_a_big_structure(A), q(B).
p(A, B)
:- something(A), r(B).
```

If the query,  $? - \text{test}(B)$ , is posed, a choice point will be created for the second clause of  $p$ . Then  $\text{create\_a\_big\_structure}(A)$  is called which sets  $A$  to refer to a very big structure. If  $A$  is not accessible through the active environment chain, this big structure is no longer needed when  $q(B)$  is called. However, from the choice point created,  $A$  is still reachable, and thus also the big structure.

How can the marking phase be modified to handle this situation? Let us examine the marking code once again. First all argument registers and all environments reachable from the current environment are marked. Then all choice points and environments reachable from these choice points are marked. But some of those variables will be reset before we backtrack to the latest choice point! The solution seems to be to reset those variables, if they are unmarked, now, before marking the choice point. (See Figure 22.)

The argument can be repeated for each choice point: unmarked variables referred to from trail entries younger than the choice point are reset before marking the choice point. This is possible to do as those variables are only reachable after backtracking, at which point they will be reset anyway.

To implement the early reset of variables,  $\text{mark\_choicepoints}$  now becomes instead

```
mark_choicepoints(cp)
  struct wam *cp;
```

```
struct valuecell **t;
t = T;
trailcells_deleted = 0;
while (cp != NULL)
{
  while (t > cp -> T)
  {
    if ((*t) -> m == FALSE)
    {
      reset(*t);
      (*t) = NULL;
      trailcells_deleted
        = trailcells_deleted + 1;
    }
    t = t - 1;
  }
  mark_environments(cp -> E,
    env_size(cp -> L));
  for each v pointing to a valuecell
  in
  cp -> A do
    mark_variable(v);
  }
  cp = cp -> B;
}
```

The procedure  $\text{reset}(x)$  sets the value cell  $x$  to unbound. Trail entries pointing to variables being reset are set to NULL as they are no longer needed. Thus, these trail entries could be discarded and the trail compacted. If this is done, entries in the trail will be moved and the corresponding pointers into the trail from the choice points have to be updated. The trail must be scanned from  $\text{trail\_low}$  to  $T$  in order to compact the

trail toward the lower addresses. Ideally, the choice points would be scanned and updated simultaneously. However, this is not possible as they are linked in the opposite direction.

One of many possible solutions to this problem is first to scan the choice points from high to low address, updating the trail pointers *T* of the choice points, and then to compact the trail going from low to high address. When scanning the choice points, the trail cells are also scanned, decrementing the counter *trailcells\_deleted* for each discarded trail entry. By subtracting the current value *trailcells\_deleted* from the *T* field of the choice point visited, that field will contain the address of the relocated trail cell; see Figure 23.

After *mark\_choicepoints* has been called, the new procedure *collect\_trail* is called, which first updates the choice points and then compacts the trail.

```
collect_trail();
{
    update_choicepoints();
    compact_trail();
}

update_choicepoints();
{
    struct choicepoint *cp;
    struct valuecell **t;
    cp = B;
    t = T;
    while (cp != NULL)
    {
        while (t > cp - T)
        {
            if ((*t) == NULL)
                trailcells_deleted
                    = trailcells_deleted - 1;
            t = t - 1;
        }
        cp - T = cp - T - trailcells_deleted;
        cp = cp - B;
    }
}

compact_trail()
{
    struct valuecell **dest, **current;
    dest = trail_low;
    for current = trail_low up to T
        if ((*current) != NULL)
        {
            (*dest) = (*current);
            dest = dest + 1;
        }
    T = dest - 1;
}
```

A method closely related to early reset called *virtual backtracking* has previously been described by Bekkers

[2], Bruynooghe [4] and Pittomvils [13]. However, in virtual backtracking the variables are not reset and the entries in the trail are not removed.

## SEGMENTED GARBAGE COLLECTION

It is possible to divide the data areas of Prolog into segments for which garbage collection can be performed independently. When a choice point is created on the stack, all currently active argument registers are copied into that choice point, fixing all structures accessible from these variables. Structures in the heap that are not garbage at that moment will remain non-garbage until the choice point is removed upon backtracking. To take advantage of this we divide the main data areas into segments. A new segment is started in the stack, heap, and trail after a choice point is placed on the stack. Only segments for which garbage collection has not been performed or segments that have been reopened upon backtracking need to be collected.

Segmented garbage collection can be implemented using one extra variable, *GC\_B*, which points to the choice point ending the oldest segment for which garbage collection has been performed. When garbage collection is performed, *GC\_B* is reset to the youngest choice point in the stack. During execution, when a choicepoint is removed, if *B* points to an older choice point than *GC\_B*, *GC\_B* is reset to *B*. Alternatively, to avoid this overhead during execution, a special bit may be set in each choice point which has taken part in a garbage collection. *GC\_B* then corresponds to the youngest choice point having this bit set. For simplicity, we ignore this optimization.

Figure 24 shows a typical situation during execution. The stack, trail, and heap are divided into two segments, new and old, by the choice point referred to by *GC\_B*.

In order to make a correct marking, we need to find all pointers going from the old segment to the new segment. The only pointers of this kind are pointers from the stack or the heap to the heap. All those pointers may be found without scanning the whole old segment since they are always recorded on the trail as shown in the figure. This is guaranteed by the fact that there is at least one choice point, the *GC\_B* choice point, which lies "between" the two cells. The unification algorithm always records such pointers on the trail, since they have to be reset upon backtracking. Thus the marking phase only needs to examine the new segment and the cells in the old segment referred from the new trail.

However—and this is the only drawback of segmented garbage collection—some of the pointers going from the old heap to the new heap would normally be reset by the "early reset" mechanism described earlier. The segmentation mechanism prevents this since all value cells in the old segment are considered equally reachable, and we cannot know whether a certain cell on the old heap is reachable from the new segment. Thus early reset is disabled for the old segment.

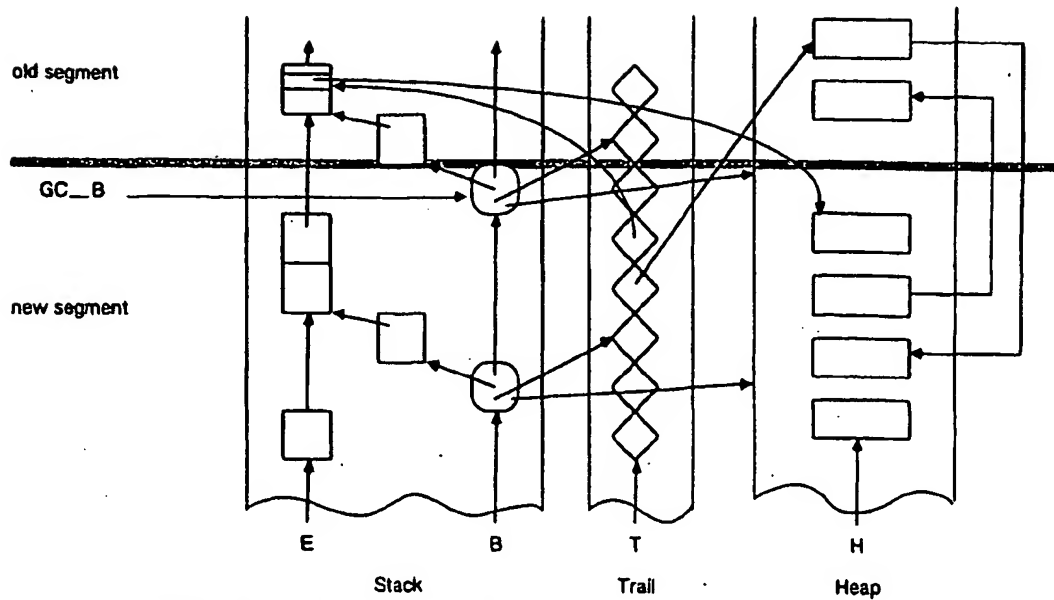


FIGURE 24. A Possible Situation Where Stack, Trail, and Heap are Divided into Two Segments

**Changes to the Marking**

A pointer  $P$  points to the heap if  $P \leq GC\_B \rightarrow H$ , it points to the old stack if  $P \leq GC\_B$ , and it points to the old trail if  $P \leq GC\_B \rightarrow T$ . We get the following modifications of the marking phase.

In `mark_environments` the while-loop instead becomes

```
while (env ≠ NULL AND env points
      to the new stack)
```

As mentioned, all pointers going from the old heap to the new heap are found on the new trail. Before calling `mark_choicepoints` in the procedure `marking` phase, the procedure `mark_trail` is called.

```
mark_trail()
{
  struct valuecell **t;
  struct valuecell temp;
  t = T;
  while (t > GC_B → T)
  {
    if ((*t) points to the old heap OR
        (*t) points to the old stack)
    { /* (**t) may point to new
       heap */
      temp = **t;
      mark_variable(&temp);
    }
    t = t + 1;
  }
}
```

In `mark_choicepoints` we instead get

```
while (cp ≠ NULL
      AND cp is not older than GC_B)
{
  while (t > cp → T)
  {
    if ((*t) does not point to the old
        heap
        AND
        (*t) does not point to the old
        stack
        AND
        (*t) → m == FALSE)
    {
      reset (*t);
      (*t) = NULL;
      trailcells_deleted =
        trailcells_deleted + 1;
    }
    t = t + 1;
  }
  mark_environments(cp → E,
    env_size(cp → L));
  for each v pointing to a valuecell
    in cp → A do mark_variable(v);
}
```

In `mark_variable` the test in the forward mode becomes

```
forward: if (current → m == TRUE) goto
backward;
current → m = TRUE;
total_marked = total_marked + 1;
if (next points to the old heap) goto
backward;
```

which reflects the idea that all objects in the old heap segment are to be considered marked and should not be touched.

#### Changes To the Compaction Phase

Since only the new heap will be compacted, only pointers into the new heap need to be considered for updating. Thus, pointers to the old heap should not be considered. The test in the procedure `push_registers` therefore becomes

```
if (A[i] — value points to the new heap) . . .
```

The procedure `sweep_trail` now only sweeps the *new* trail, putting all entries pointing to the new heap into the relocation chain as before (case a in Figure 25). Trail entries pointing to the old segment do not need to be updated as the old segment will not be moved by compaction (cases b, c, and d). However, if the old heap or stack cell refers to the new heap (cases b and d), that cell needs to be updated.

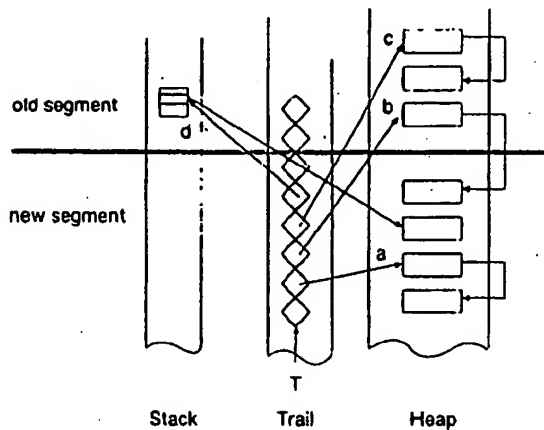


FIGURE 25. Possible Situations Encountered While Sweeping the Trail

A modified version of `sweep_trail`, which takes care of those special cases, now looks like this:

```
sweep_trail()
{
  struct valuecell **current;
  for current = T down to GC_B → T + 1
    if ((*current) is a new heap
        pointer)
      into_relocation_chain
        ((*current), current);
    else if ((*current) is an old heap
             or stack pointer AND
             (*current) is a new heap
             pointer)
      into_relocation_chain
        ((*current),
          (*current));
}
```

The modifications needed for `sweep_environments` and `sweep_choicepoints` are quite similar. By adding a test env points to the new stack and cp points to the new stack, respectively, to the main while-loops, only the new stack will be scanned. The tests for references to the heap are changed into tests for references to the *new* heap.

#### DISCUSSION

It is interesting to compare our marking algorithm to Thorelli's [12], which also uses a pointer reversal technique not needing any extra storage for the traversal besides one cell per object. This extra cell contains a counter indicating how far the object has been marked. If the largest object contains  $n$  cells, the counter must contain at least  $\lceil \log_2 n \rceil$  bits. If many objects are close to the maximum object size, Thorelli's method clearly becomes superior to our method where two bits are needed for each cell. However, most objects in Prolog are much smaller than the largest one. The main reason for not using Thorelli's approach is that it does not seem possible to extend his scheme to mark just parts of an object.

In one of the first described garbage collection algorithms specifically for Prolog by Warren [15], the outer unmarked parts of an object may be freed, but not the inner unmarked parts surrounded by marked cells. This limitation seems inherent in implementations based on "structure sharing." A source of inspiration for our work has been the series of publications by Bruynooghe [3-5] leading to Pittomvils [13], where the ideas of virtual backtracking [2] and segmented garbage collection [9] have been incorporated.

The main novel result of this article is the marking algorithm, which needs only two bits per word and no extra stack space. It also has the capability of marking only the parts of a structure that are reachable. Another novelty is the concept of "early reset of variables," instead of virtual backtracking. By resetting a variable ("early reset"), instead of indicating that what it points to should not be marked ("virtual backtracking"), we are able to compact the trail, as some trail entries are no longer needed. The paper gives a fairly detailed description of how to implement the various parts of the garbage collection algorithm, including how to adapt it to segmented garbage collection. The compaction algorithm presented here is proportional to the size of the heap, not to the number of reachable objects. Given a program which generates a lot of garbage, a significant amount of time may be spent in the compaction phase. Adding a new phase inserted between marking and compaction which links the marked objects using the unmarked objects as link-nodes, would enable the compaction phase to just scan the marked cells. Such an intermediate phase can be made proportional to  $n \log n$ , where  $n$  is the number of marked cells [11].

The question of completeness naturally arises, that is, does the algorithm find and deallocate all garbage. For deterministic languages the question is easy to answer,

but for a nondeterministic language it becomes more complex since the machine contains many frozen states. Without giving a proof we claim the garbage collection algorithm presented (with "early reset" but without segmentation) is complete in the following sense: Assume that we had a WAM machine that could "fork" itself at each choice point. Each machine would then execute deterministically, not having any choice points at all. A set of such machines corresponds to the state of an ordinary WAM machine having choice-points. The objects reachable in all these machines correspond exactly to those objects we consider reachable in our algorithm.

This does not mean, however, that we have reclaimed all possible storage. The "single assignment" property of Prolog makes more semantical optimizations possible. For instance, chains of variables may be collapsed, making it possible to deallocate the intermediate cells and thus also speed up execution later.

Another storage optimization would be to find objects of exactly the same structure and collapse all these objects into one canonical representative. By having a special "canonical" bit in the pointer to those objects, the unification of two such objects just becomes a pointer comparison. Finding all equal objects is possibly too time consuming for this optimization to be worthwhile, but this remains to be investigated.

**Acknowledgments.** We are grateful for the valuable comments given by Peter Sheridan, Khayri Mohamed Ali, Andrzej Ciepielewski, Gunnar Blomberg and the anonymous referees on earlier versions of this article. In particular we would like to thank Göran Båge for finding the "last bugs" while implementing the algorithm.

#### REFERENCES

1. Barklund, J., and Millroth, H. Garbage cut for garbage collection of iterative Prolog programs. In *Proceedings of the 1986 Symposium on Logic Programming* (Salt Lake City, Utah).
2. Bekkers, Y., Canet, B., Ridoix, O., and Ungaro, L. A short note on garbage collection in Prolog interpreters. *Logic Programming Newsletter*, no. 5 (Winter 83/84).
3. Bruynooghe, M. A note on garbage collection in Prolog interpreters. In *Proceedings of the First International Logic Programming Conference*, 1982, pp. 52-55.
4. Bruynooghe, M. Garbage collection in Prolog interpreters. In J. Campbell (Ed.), *Implementations of PROLOG*. Ellis Horwood, Chichester, England 1984, pp. 259-267.
5. Bruynooghe, M. The memory management of Prolog implementations. In K.L. Clark and S.-Å. Tärnlund (Eds.), *Logic Programming*. Academic Press, New York, 1982, pp. 82-98.
6. Carlsson, M. Compilation for Tricia and its abstract machine. Tech. Rep. 35, 1986, UPMail, Uppsala University.
7. Gabriel, J., Lindholm, T., Lusk, E.L., and Overbeek, R.A. Tutorial on the Warren abstract machine for computational logic. Tech. Rep. ANL-84-84, Argonne National Lab., Argonne, Ill.
8. Lieberman, H., and Hewitt, C. A real time garbage collector based on the life time of objects. *Commun. ACM* 26, 6 (June 1983), 419-429.
9. Morris, F.L. A time and space efficient garbage compaction algorithm. *Commun. ACM* 21, 8 (Aug. 1978), 662-665.
10. Sahlin, D. Garbage collection using the reset information and making tests deterministic using the reset information. SICS working document. SICS, Kista, Sweden, Mar. 1986.
11. Sahlin, D. Making the garbage collection independent of the amount of garbage. Res. Rep. R87008, SICS, Kista, Sweden, ISSN 0283-3638.
12. Thorelli, L.-E. Marking algorithms. *BIT* 12, 4 (1972), 555-568.
13. Pittonvils, E., Bruynooghe, M., Willems, Y.D. Towards a real time garbage collector for Prolog. In *Proceedings of the Symposium on Logic Programming*, 1985, pp. 185-198.
14. Warren, D.H.D. An abstract Prolog instruction set. Tech. Note 309, SRI International, Menlo Park, Calif., Oct. 1983.
15. Warren, D.H.D. Implementing Prolog—Compiling predicate logic programs. Rep. 39 and 40, University of Edinburgh, Department of Artificial Intelligence, Edinburgh, Scotland, May 1977.
16. Warren, D.S. The runtime environment for a Prolog compiler using a copy algorithm. Tech. Rep. 83/052, SUNY at Stony Brook, N.Y., 1983. Major revision, March 1984.

Revised 1/88; accepted 2/88

Authors' Present Addresses: K. Appleby, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598; M. Carlsson, Seif Haridi, and D. Sahlin, SICS, P.O. Box 1263, S-164 28 Kista, Sweden.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

## ACM Algorithms

Collected Algorithms from ACM (CALGO) now includes quarterly issues of complete algorithm listings on microfiche as part of the regular CALGO supplement service.

The ACM Algorithms Distribution Service now offers microfiche containing complete listings of ACM algorithms, and also offers compilations of algorithms on tape as a substitute for tapes containing single algorithms. The fiche and tape compilations are available by quarter and by year. Tape compilations covering five years will also be available.

To subscribe to CALGO, request an order form and a free ACM Publications Catalog from the ACM Subscription Department, Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036. To order from the ACM Algorithms Distributions Service, refer to the order form that appears in every issue of ACM Transactions on Mathematical Software.

